Introduction to MIPS Programming

Dr. Edward Nava
ejnava@unm.edu

MIPS Instruction Set

See the MIPS Instruction Set Reference Manual for a detailed description of every instruction.

- **Arithmetic, Logic, and Shifting Instructions**
- **Conditional Branch Instructions**
- **Load and Store Instructions**
- **Function Call Instructions**

An Example

Let us suppose that an assembly language programmer wants to add the contents of register $a1 to the contents of register $s1, and to place the result in register $v1. The assembly language instruction to accomplish this is:

add $v1, $a1, $s1

The equivalent pseudocode statement is:

$v1 = $a1 + $s1

Quick Reference – Appendix A

Integer Instruction Set

Name	Syntax		Space/Time
Add:	add	Rd, Rs, Rt	1/1
Add Immediate:	addi	Rt, Rs, Imm	1/1
Add Immediate Unsigned:	addiu	Rt, Rs, Imm	1/1
Add Unsigned:	addu	Rd, Rs, Rt	1/1
And:	and	Rd, Rs, Rt	1/1
And Immediate:	andi	Rt, Rs, Imm	1/1
Branch if Equal:	beq	Rs, Rt, Label	1/1
Branch if Greater Than or Equal to Zero:	bgez	Rs, Label	1/1
Branch if Greater Than or Equal to Zero and Link:	bgezal	Rs, Label	1/1
Branch if Greater Than Zero:	bgtz	Rs, Label	1/1
Branch if Less Than or Equal to Zero:	blez	Rs, Label	1/1
Branch if Less Than Zero and Link:	bltzal	Rs, Label	1/1
Branch if Less Than Zero:	bltz	Rs, Label	1/1
Branch if Not Equal:	bne	Rs, Rt, Label	1/1
Divide:	div	Rs, Rt	1/38
Divide Unsigned:	divu	Rs, Rt	1/38
Jump:	j	Label	1/1
Jump and Link:	jal	Label	1/1
Jump and Link Register:	jalr	Rd, Rs	1/1
Jump Register:	jr	Rs	1/1

Integer Instruction Set

Name	Syntax		Space/Time
Load Byte:	lb	Rt, offset(Rs)	1/1
Load Byte Unsigned:	lbu	Rt, offset(Rs)	1/1
Load Halfword:	lh	Rt, offset(Rs)	1/1
Load Halfword Unsigned:	lhu	Rt, offset(Rs)	1/1
Load Upper Immediate:	lui	Rt, Imm	1/1
Load Word:	lw	Rt, offset(Rs)	1/1
Load Word Left:	lwl	Rt, offset(Rs)	1/1
Load Word Right:	lwr	Rt, offset(Rs)	1/1
Move From Coprocessor 0	mfc0	Rd, Cs	1/1
Move From High:	mfhi	Rd	1/1
Move From Low:	mflo	Rd	1/1
Move To Coprocessor 0	mtc0	Rt, Cd	1/1
Move to High:	mthi	Rs	1/1
Move to Low:	mtlo	Rs	1/1
Multiply:	mult	Rs, Rt	1/32
Multiply Unsigned:	multu	Rs, Rt	1/32
NOR:	nor	Rd, Rs, Rt	1/1
OR:	or	Rd, Rs, Rt	1/1
OR Immediate:	ori	Rt, Rs, Imm	1/1
Return From Exception:	rfe		1/1
Store Byte:	sb	Rt, offset(Rs)	1/1
Store Halfword:	sh	Rt, offset(Rs)	1/1
Shift Left Logical:	sll	Rd, Rt, sa	1/1
Shift Left Logical Variable:	sllv	Rd, Rt, Rs	1/1

Integer Instruction Set

Name	Syntax		Space/Time
Set on Less Than:	slt	Rd, Rt, Rs	1/1
Set on Less Than Immediate:	slti	Rt, Rs, Imm	1/1
Set on Less Than Immediate Unsigned:	sltiu	Rt, Rs, Imm	1/1
Set on Less Than Unsigned:	sltu	Rd, Rt, Rs	1/1
Shift Right Arithmetic:	sra	Rd, Rt, sa	1/1
Shift Right Arithmetic Variable:	srav	Rd, Rt, Rs	1/1
Shift Right Logical:	srl	Rd, Rt, sa	1/1
Shift Right Logical Variable:	srlv	Rd, Rt, Rs	1/1
Subtract:	sub	Rd, Rs, Rt	1/1
Subtract Unsigned:	subu	Rd, Rs, Rt	1/1
Store Word:	sw	Rt, offset(Rs)	1/1
Store Word Left:	swl	Rt, offset(Rs)	1/1
Store Right:	swr	Rt, offset(Rs)	1/1
System Call:	syscall		1/1
Exclusive OR:	xor	Rd, Rs, Rt	1/1
Exclusive OR Immediate:	xori	Rt, Rs, Imm	1/1

Macro Instructions

- **Load Address** `la $s0, table`
- **Load Immediate** `li $v0, 10`
- **Move** `mov $t8, $sp`
- **Multiply** `mul $t2, $a0, $a1`
- **Divide** `div $s1, $v1, $t7`
- **Remainder** `rem $s2, $v1, $t7`
- **Negate** `neg $s0, $s0`

Integer Multiplication

The multiply instruction "**mult"** multiplies two 32-bit binary values and produces a 64-bit product which is stored in two special registers named High and Low. In this case, the destination for the result is implicitly understood.

Register High will be loaded with the upper 32-bits of the product and register Low will be loaded with the lower 32-bits of the product.

To move a copy of the value in the High register to the register file we use the instruction **mfhi** and to move a copy of the value in the Low register to the register file we use the instruction **mflo**.

The following code segment shows how the 64-bit product of $a1 times $s1 can be moved into $v0 and $v1:

```
mult    $a1, $s1
mfhi    $v0
mflo    $v1
```

Integer Divide

The following *divide instruction* divides the 32-bit binary value in register $a1 by the 32-bit value in register $s1. The *quotient* is stored in the Low register and the *remainder* is stored in the High register.

The following code segment shows how the quotient is moved into $v0 and the remainder is moved into $v1:

```
div     $a1, $s1
mflo    $v0
mfhi    $v1
```

In the MIPS architecture, division by zero is undefined. If it is possible that the divisor could be zero, then it is the programmers responsibility to test for this condition and to provide code to handle this special situation.

Control Structures

There are instructions to implement *control structures* such as:
“if ... then ... else ...”

Let us suppose that if the contents of register $s6 is less than zero, in other words negative, we want to branch to a location in the program labeled “Quit.” Otherwise (else) we want to decrement the contents of register $s6.

if ($s6 >= 0) then ($s6 = $s6 – 1) else goto Quit

The assembly language instructions to accomplish this are:

```
bltz    $s6, Quit
addi    $s6, $s6, -1
```

An Example MIPS Assembly Language Program to find the sum of the integers from 1 to 99

```
Label       Op-Code  Dest.  S1,   S2        Comments
            move     $a0, $0                # $a0 = 0
            li       $t0, 99                # $t0 = 99
loop:
            add      $a0, $a0, $t0          # $a0 = $a0 + $t0
            addi     $t0, $t0, -1           # $t0 = $t0 - 1
            bnez     $t0, loop              # if ($t0  != zero) branch loop
            li       $v0, 1                 # Print the value in $a0
            syscall
            li       $v0, 10                # Terminate Program Run
            syscall
```

Memory Addressing Modes

The MIPS architecture is a ***Load/Store*** *architecture*, which means the only instructions that access main memory are the load and store instructions.

Only one *addressing mode* is implemented in the hardware.

The addressing mode is referred to as *base address plus displacement.*

Memory Addressing Details

- A *load instruction* accesses a value from memory and places a copy of the value found in memory in the register file. For example, the instruction:

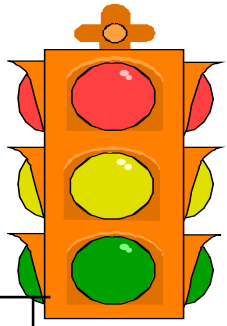
  lw $s1, 8($a0)

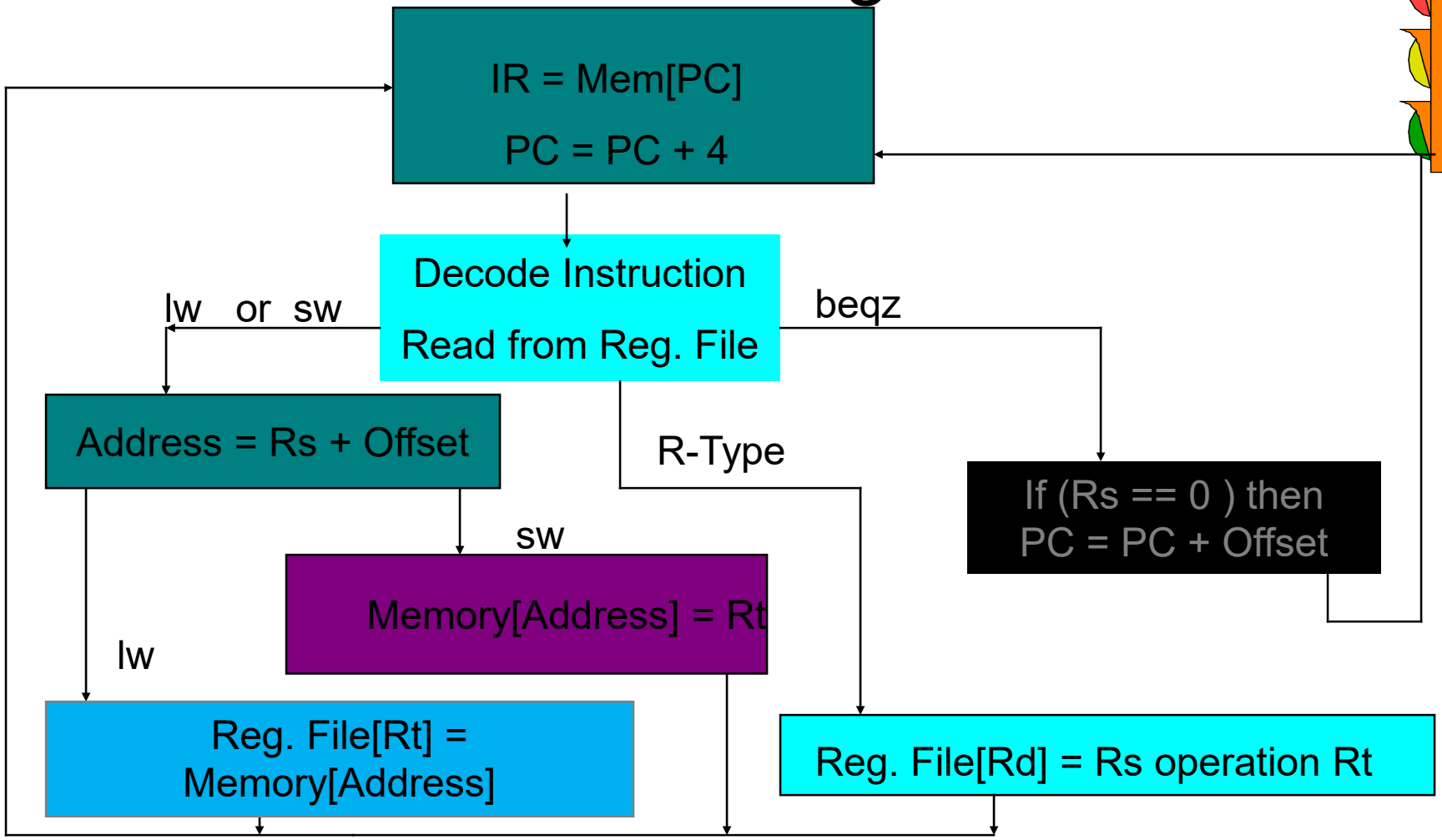
  computes the *effective address* of the memory location to be accessed by adding together the contents of register $a0 (the *base address*) and the constant value eight (the *displacement*).

- A copy of the value accessed from memory at the effective address is loaded into register $s1. The equivalent *pseudocode* statement would be:

  $s1 = Mem[$a0 + 8]

A Register Transfer Description of the Control Logic





Memory Addressing Syntax

The syntax of the assembly language load instruction is somewhat confusing.

lw $s1, 8($a0)

Memory Addressing Example

The following is an example of a “Store Word” instruction:

sw $s1, 12($a0)

When the hardware executes this instruction it will compute the effective address of the destination memory location by adding together the contents of register $a0 and the constant value 12. A copy of the contents of register $s1 is stored in memory at the effective address.

The equivalent *pseudocode* statement would be:

Mem[$a0 + 12] = $s1

From the point of view of an assembly language programmer, memory can be thought of as a very long linear array of locations where binary codes are stored. An effective address is a pointer to some location in this array.